APPENDIX A.I

Source code file named compound.pl.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   File    : compound.pl
%   Primary Authors  : David Martin, Adam Cheyer
%   Purpose : Provides handling of compound goals by the facilitator.
%
%   -------------------------------------------------------------------------
%   Unpublished-rights reserved under the copyright laws of the United States.
%
%
%   Unpublished Copyright (c) 1998, SRI International.
%   "Open Agent Architecture" and "OAA" are Trademarks of SRI International.
%   -------------------------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This is just here so this file can be compiled separately (but its
% official declaration is in oaa.pl):
:- op(599,yfx,::).

:- dynamic
    binding_num/1,
    ks_num/1,
    multiple_continuation/7
.

% This file is loaded by facilitator code, and thus no
% module imports are needed here.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OVERVIEW
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*\

These facilitator routines support the use of compound "ICL goals".
An ICLGoal is of the form Sources:Goal::Params, where both Sources and
Params are optional.  Each subgoal of ICLGoal is also of that form.

When an agent calls solve/2, it may specify an ICL goal which is
"incomplete"; that is, ambiguous as to which agents are to solve the
various subgoals.  The facilitator then completes the ICL goal, if
necessary, and executes it.  Execution involves having all the
subgoals solved by the appropriate agents, assembling the solutions,
and returning them to the requesting agent.

If a agent wants to construct a complete ICL goal, and is willing to
guarantee that it's complete and that all solvers mentioned in it are
currently valid, then that agent (usually a "meta-agent") may call
execute_goal directly.  @@ We haven't yet provided library calls for
this.

IMPORTANT NOTE: : has higher precedence than ::.  This means that
a:b::c will unify with X:Y and X:Y::Z, but NOT with Y::Z.

Wherever a Sources field appears, it may be any of the following:
    built_in
    facilitator
```

```
    parent
    KS
    [KS1, KS2, ...]
```

'built_in' isn't normally specified by a requesting agent - although there's no harm in doing so - but is used internally by the facilitator. KS, KS1, KS2, etc. may be either the name or address of an agent (client or facilitator). 'facilitator' or 'parent' may also appear in a list of KS's. If Sources is an empty list or a var, it is handled just as if there were no Sources field, in which case the facilitator determines what sources are relevant.

Note that when an ICL goal includes a Sources field, there should not be Sources fields for any of its subgoals. If there are, they will be ignored. (@@Need to make sure this works ok.) However, Params fields may be usefully nested within goals that have Params fields. Certain nested parameters, such as solution_limit/1, can be used by the solving agent.

If an ICL goal has parameters, some of them are "inherited" by subgoals. If there's a conflicting parameter on a subgoal, however, it overrides an inherited parameter.

PARAMETERS
----

address(+A) [embedded or global] - Used precisely as if A: prefixes the relevant goal.

get_address(-S) [embedded] - bind S to indicate who provided the solution. Solver identities will be given as numeric ids. Currently only works when attached to non-compound (sub)goals.

get_address(-S) [global] - bind S to indicate all sources that were queried in finding solutions (even if they returned none).

*/

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GOAL COMPLETION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

/*\

complete_goal(RequestingKS, Goal, GlobalParams, CompletedGoal).

complete_goal takes in an ICL goal and produces a "complete ICL goal" (sometimes known as a "plan", but I think we'll reserve that term for future developments). The goal and the complete goal have precisely the same variables - but are not necessarily unifiable.

*/

```
complete_goal(RequestingKS, Goal, GlobalParams, CompletedGoal) :-
    complete_addressing(RequestingKS, Goal, GlobalParams, AddressedGoal),
    complete_concurrency(AddressedGoal, CompletedGoal).
```

```
/*\

complete_addressing(+RequestingKS, +ICLGoal, +GlobalParams, -AddressedGoal).

AddressedGoal has more-or-less the same form as ICLGoal, but possibly
with some regrouping of subgoals, and the addition of Sources fields
to ICLGoal or its subgoals.  The idea is that AddressedGoal contains
complete information as to where its various subgoals are to be sent,
so that no further analysis is needed.  Any regrouping of subgoals is
done as an optimization.  AddressedGoal shares all variables with
ICLGoal.

@@What other operators (e.g., negation) might we want to support?

\*/

complete_addressing(RequestingKS, ICLGoal, GlobalParams, AddressedGoal) :-
    % @@ verify_params(GlobalParams, global, Verified),
    complete_sources(RequestingKS, ICLGoal, GlobalParams,
                 AddressedGoalWithParamsEverywhere),
    % @@Here, propagate params, instantiate address request in GlobalParams. ?
    remove_empty_params(AddressedGoalWithParamsEverywhere, AddressedGoal).

/*\

complete_sources(+RequestingKS, +ICLGoal, +GlobalParams, -AddressedGoal).

Ensures that every subgoal is explicitly covered by one or more
sources.  Determines the largest subgoals that can be "chunked"; that
is, grouped together for submission to a source.

In the process, every goal acquires a Params field (wherever there was
no Params field before, the empty list is added).  This is done just
to make the definition of complete_sources more readable.

\*/

    % Here we assume that the goal-writer didn't really mean to put a var,
    % because it's not meaningful to do so:
complete_sources(KS, Sources:Goal, GlobalParams, AddressedGoal) :-
    var(Sources),
    !,
    complete_sources(KS, Goal, GlobalParams, AddressedGoal).

/*
    ( AddressedGoal = A:_ ->
      Sources = A
    | otherwise ->
      findall(A, sub_term(A:_, AddressedGoal), SubSources),
      % @@More work needed here:
      Sources = SubSources
    ).
*/

    % Here we assume that the goal-writer didn't really mean to put [],
    % because it's not meaningful to do so:
```

```
complete_sources(KS, []:Goal, GlobalParams, AddressedGoal) :-
    !,
    complete_sources(KS, Goal, GlobalParams, AddressedGoal).

    % Sources and Params already specified; we're done:
    % @@But let's verify the sources are valid!
complete_sources(_KS, Sources:Goal::Params, _GlobalParams,
             Sources:Goal::Params) :-
    !.

    % Sources already specified; add empty Params list:
complete_sources(_KS, Sources:Goal, _GlobalParams, Sources:Goal::[]) :-
    !.

    % Sure, we'll continue to support an address in Params or GlobalParams:
complete_sources(KS, Goal::Params, GlobalParams, AddressedGoal) :-
    % @@ verify_params(...),
    ( memberchk(address(Sources), Params) ;
      memberchk(address(Sources), GlobalParams) ),
    \+ var(Sources),
    !,
    complete_sources(KS, Sources:Goal::Params, GlobalParams, AddressedGoal).

    % No Sources or Params specified; add empty Params list before
    % proceeding:
complete_sources(KS, Goal, GlobalParams, AddressedGoal) :-
    \+ (Goal = _::_),
    !,
    complete_sources(KS, Goal::[], GlobalParams, AddressedGoal).

    % Here we get down to the real work: determining solvers and
    % chunking of subgoals:

complete_sources(KS, (\+ Goal1)::Params, GlobalParams, AddressedGoal) :-
    !,
    oaa_Name(Facilitator),
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
      % If S1 is a SINGLE source, it's OK to send the negation to the source.
      % This case also works if S1 == built_in.
    ( (AddressedGoal1 = [S1]:G1::P1,
       S1 \== Facilitator,
       S1 \== facilitator) ->
        AddressedGoal = S1:((\+ G1)::P1)::Params
    | otherwise ->
        AddressedGoal = (\+ AddressedGoal1::Params)
    ).

complete_sources(KS, (Goal1, Goal2, Goal3)::Params, GlobalParams,
             AddressedGoal) :-
    % This clause is needed because we want built_in pred's to be grouped
    % with what comes before, not after.
    !,
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
    complete_sources(KS, Goal2, GlobalParams, AddressedGoal2),
    complete_sources(KS, Goal3, GlobalParams, AddressedGoal3),
    ( (AddressedGoal1 = S1:G1::P1,
       AddressedGoal2 = S2:G2::P2,
```

```
        AddressedGoal3 = S3:G3::P3,
        chunkable_sources([S1, S2, S3], Sources),
        compatible_params([P1, P2, P3])) ->
        AddressedGoal = Sources:(G1::P1, G2::P2, G3::P3)::Params
    | (AddressedGoal1 = S1:G1::P1,
        AddressedGoal2 = S2:G2::P2,
        AddressedGoal3 = (S3A:G3A::P3A, Goal3B)::P3,
        % Goal3B may or may not begin with Source:.  icl_GoalComponents
        % deals with the precedence issues.
        icl_GoalComponents(Goal3B, _, G3B, P3B),
        chunkable_sources([S1, S2, S3A], Sources),
        append(P3A, P3, NewP3A),
        append(P3B, P3, NewP3B),
        compatible_params([P1, P2, NewP3A])) ->
        AddressedGoal = (Sources:(G1::P1, G2::P2, G3A::NewP3A)::[],
                         G3B::NewP3B)::Params
    | (AddressedGoal1 = S1:G1::P1,
        AddressedGoal2 = S2:G2::P2,
        chunkable_sources(S1, S2, Sources),
        compatible_params([P1, P2])) ->
        AddressedGoal = (Sources:(G1::P1, G2::P2)::[], AddressedGoal3)::Params
    | (AddressedGoal2 = S2:G2::P2,
        AddressedGoal3 = S3:G3::P3,
        chunkable_sources(S2, S3, Sources),
        compatible_params([P2, P3])) ->
        AddressedGoal = (AddressedGoal1, Sources:(G2::P2, G3::P3)::[])::Params
    | (AddressedGoal2 = S2:G2::P2,
        AddressedGoal3 = (S3A:G3A::P3A, Goal3B)::P3,
        icl_GoalComponents(Goal3B, _, G3B, P3B),
        chunkable_sources([S2, S3A], Sources),
        append(P3A, P3, NewP3A),
        append(P3B, P3, NewP3B),
        compatible_params([P2, NewP3A])) ->
        AddressedGoal = (AddressedGoal1, Sources:(G2::P2, G3A::NewP3A)::[],
                         G3B:NewP3B)::Params
    | otherwise ->
        AddressedGoal =
            (AddressedGoal1, AddressedGoal2, AddressedGoal3)::Params
    ).
complete_sources(KS, (Goal1, Goal2)::Params, GlobalParams, AddressedGoal) :-
    !,
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
    complete_sources(KS, Goal2, GlobalParams, AddressedGoal2),
    ( (AddressedGoal1 = S1:G1::P1,
        AddressedGoal2 = S2:G2::P2,
        chunkable_sources(S1, S2, Sources),
        compatible_params([P1, P2])) ->
        AddressedGoal = Sources:(G1::P1, G2::P2)::Params
    | otherwise ->
        AddressedGoal = (AddressedGoal1, AddressedGoal2)::Params
    ).
    % Note: this clause must precede that for disjunction.
complete_sources(KS, (Goal1 -> Goal2 ; Goal3)::Params, GlobalParams,
            AddressedGoal) :-
    !,
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
    complete_sources(KS, Goal2, GlobalParams, AddressedGoal2),
```

```
    complete_sources(KS, Goal3, GlobalParams, AddressedGoal3),
    ( (AddressedGoal1 = S1:G1::P1,
       AddressedGoal2 = S2:G2::P2,
       AddressedGoal3 = S3:G3::P3,
       chunkable_sources([S1, S2, S3], Sources),
       compatible_params([P1, P2, P3])) ->
       AddressedGoal = Sources:(G1::P1 -> G2::P2 | G3::P3)::Params
    | otherwise ->
       AddressedGoal =
         (AddressedGoal1 -> AddressedGoal2 | AddressedGoal3)::Params
    ).
complete_sources(KS, (Goal1 -> Goal2)::Params, GlobalParams, AddressedGoal) :-
    !,
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
    complete_sources(KS, Goal2, GlobalParams, AddressedGoal2),
    ( (AddressedGoal1 = S1:G1::P1,
       AddressedGoal2 = S2:G2::P2,
       chunkable_sources([S1, S2], Sources),
       compatible_params([P1, P2])) ->
       AddressedGoal = Sources:(G1::P1 -> G2::P2)::Params
    | otherwise ->
       AddressedGoal =
         (AddressedGoal1 -> AddressedGoal2)::Params
    ).
complete_sources(KS, (Goal1 ; Goal2)::Params, GlobalParams, AddressedGoal) :-
    !,
    complete_sources(KS, Goal1, GlobalParams, AddressedGoal1),
    complete_sources(KS, Goal2, GlobalParams, AddressedGoal2),
    ( (AddressedGoal1 = S1:G1::P1,
       AddressedGoal2 = S2:G2::P2,
       chunkable_sources(S1, S2, Sources),
       compatible_params([P1, P2])) ->
       AddressedGoal = Sources:(G1::P1; G2::P2)::Params
    | otherwise ->
       AddressedGoal = (AddressedGoal1; AddressedGoal2)::Params
    ).
    % To be complete, we will allow for this nonstandard goal form:
complete_sources(KS, Goal::Params1::Params2, GlobalParams,
             AddressedGoal::Params2) :-
    !,
    complete_sources(KS, Goal::Params1, GlobalParams, AddressedGoal).
complete_sources(_KS, Goal::Params, _GlobalParams, built_in:Goal::Params) :-
    icl_BuiltIn(Goal),
    !.
    % Here, finally, we determine the agents (or parent facilitator) that
    % can solve a non-compound Goal:
complete_sources(KS, Goal, GlobalParams, Sources:Goal) :-
    sources_for_goal(KS, Goal, GlobalParams, Sources).

remove_empty_params(Addr:Goal::[], Addr:NewGoal) :-
    !,
    remove_empty_params(Goal, NewGoal).
remove_empty_params(Addr:Goal::Params, Addr:NewGoal::Params) :-
    !,
    remove_empty_params(Goal, NewGoal).
remove_empty_params(Goal::[], NewGoal) :-
    !,
```

```
    remove_empty_params(Goal, NewGoal).
remove_empty_params(Goal::Params, NewGoal::Params) :-
    !,
    remove_empty_params(Goal, NewGoal).
remove_empty_params(Sources:Goal, Sources:NewGoal) :-
    !,
    remove_empty_params(Goal, NewGoal).
remove_empty_params((\+ Goal)::[], (\+ NewGoal)) :-
    !,
    remove_empty_params(Goal, NewGoal).
remove_empty_params((Goal1, Goal2), (NewGoal1, NewGoal2)) :-
    !,
    remove_empty_params(Goal1, NewGoal1),
    remove_empty_params(Goal2, NewGoal2).
remove_empty_params((Goal1 ; Goal2), (NewGoal1 ; NewGoal2)) :-
    !,
    remove_empty_params(Goal1, NewGoal1),
    remove_empty_params(Goal2, NewGoal2).
remove_empty_params((Goal1 -> Goal2), (NewGoal1 -> NewGoal2)) :-
    !,
    remove_empty_params(Goal1, NewGoal1),
    remove_empty_params(Goal2, NewGoal2).
    % Primitive (non-compound) goal:
remove_empty_params(Goal, Goal).

remove_addresses(_Sources:Goal, NewGoal) :-
    !,
    remove_addresses(Goal, NewGoal).
remove_addresses((Goal1, Goal2), (NewGoal1, NewGoal2)) :-
    !,
    remove_addresses(Goal1, NewGoal1),
    remove_addresses(Goal2, NewGoal2).
remove_addresses((Goal1 ; Goal2), (NewGoal1 ; NewGoal2)) :-
    !,
    remove_addresses(Goal1, NewGoal1),
    remove_addresses(Goal2, NewGoal2).
remove_addresses((Goal1 -> Goal2), (NewGoal1 -> NewGoal2)) :-
    !,
    remove_addresses(Goal1, NewGoal1),
    remove_addresses(Goal2, NewGoal2).
    % Primitive (non-compound) goal:
remove_addresses(Goal, Goal).

/*\

chunkable_sources(+Sources1, +Sources2, -Sources).

Each argument is either: a single KS name (or numeric id); a list of
KS names (where 'facilitator' or 'parent' also count as KS
names), or the atom 'built_in'.  (Empty list is OK.)

Sources1 gives the sources that can solve some goal, Sources2
gives the sources that can solve some other goal, and if this
pred. succeeds, Sources gives a set of sources that can solve
both together.

NOTES ON CHUNKING:
```

```
%1 A chunk is a sub-goal SG of a Goal such that
  (1) There is a nonempty set S of client agents each of which can solve
the entire chunk (that is, every predicate in the chunk is either an
icl_BuiltIn or one of the agent's solvables), and
  (2) Performing the subgoal as (ks1:SQ ; ks2:SQ ; ... ; ksN:SQ), where
ks1 ... ksN are all the agents in S, does not in any way violate the
intended semantics of the overall Goal.

NOTE:  chunking is done "conservatively", so as to preserve Prolog
semantics.  So, for example, the following Goal:
    (a(1), b(2)),
where a and b are both solvable by ks1 and ks2, will be chunked as
follows:
    chunk(a(1), [ks1, ks2]), chunk(b(2), [ks1, ks2])
which amounts to no chunking at all, instead of
    chunk((a(1), b(2)), [ks1, ks2]).

The former results in execution
    (ks1:a(1) ; ks2:a2), (ks1:b(2) ; ks2:b(2))
whereas the latter would result in execution
    ks1:(a(1), b(2)) ; ks2:(a(1), b(2))
We might want to explore under what conditions more extensive chunking
can be done.

\*/

    % This just allows for single sources, not in a list:
chunkable_sources(Source1, Source2, Sources) :-
    ( atomic(Source1) ->
      S1 = [Source1]
    | otherwise ->
      S1 = Source1
    ),
    ( atomic(Source2) ->
      S2 = [Source2]
    | otherwise ->
      S2 = Source2
    ),
    chunkable_srcs(S1, S2, Sources).

chunkable_srcs(built_in, Sources, Sources) :-
    % at least one element:
    Sources = [_ | _],
    !.
chunkable_srcs(Sources, built_in, Sources) :-
    Sources = [_ | _],
    !.
chunkable_srcs([], [], []) :-
    !.
chunkable_srcs([Source], [Source], [Source]) :-
    !.
chunkable_srcs([Source1], [Source2], [Source1]) :-
    ( number(Source1), atom(Source2) ;
      number(Source2), atom(Source1) ),
    !,
    find_address(Source1, Source),
    find_address(Source2, Source).
```

```
    % chunkable_sources(+SourcesIn, -SourcesOut).
    %   Does the same as chunkable_sources/3, but allows for a list
    % of sources (length >= 1) as arg 1.

chunkable_sources([Sources], Sources).
chunkable_sources([Sources1, Sources2 | RestSources], SourcesOut) :-
    chunkable_sources(Sources1, Sources2, SourcesTemp),
    chunkable_sources([SourcesTemp | RestSources], SourcesOut).

    % compatible_params(+ParamLists).
    %   ParamLists is a list of 2 or more ParamLists.  This predicate
    % succeeds IFF the ParamLists are compatible for purposes of
    % chunking.
compatible_params(_).

    % sources_for_goal(+RequestingKS, +Goal, +Params, -Sources).
    % @@ Here, depending on how the treatment of multiple facilitators evolves,
    % we may need to revisit the default use of the facilitator.

sources_for_goal(RequestingKS, ICLGoal, GlobalParams, Sources) :-
    icl_GoalComponents(ICLGoal, _, Goal, Params),
    append(Params, GlobalParams, AllParams),
    findall(SomeKS,
          choose_ks_for_goal(RequestingKS,Goal,_,AllParams,SomeKS,_),
          KSList),
    ( KSList = [] ->
      % @@Determine if there's a parent facilitator that can handle
      % the goal.  This needs work; probably should have a local record
      % of what the parent can handle.
      find_level(AllParams, Level, _NewParams),
      ( (on_exception(_, com:com_GetInfo(parent, fac_id(ParentBB)), fail), Level
> 0) ->
          Sources = [ParentBB]
      | otherwise ->
          Sources = []
      )
    | otherwise ->
        Sources = KSList
    ).

    % If Sources is bound, VERIFIES that all the Sources can be used
    % on the ICLGoal.  If var(Sources), finds all the Sources that can
    % be used.

% sources_for_compound_goal(RKS, ICLGoal, GlobalParams, Sources) :-

/*\

complete_concurrency(+Goal, -ConcurrentGoal).

TBD.

\*/

complete_concurrency(Goal, Goal).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GOAL EXECUTION: TOP LEVEL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*\
    execute_goal(+RequestingKS, +OrigGoal, +OrigParams, +CompleteGoal).

OrigGoal are OrigParams are exactly as submitted by some client agent
(RequestingKS).  CompleteGoal is the rewriting of OrigGoal that
ensures complete addressing.  OrigGoal and ICLGoal contain precisely
the same var's.

See global comments near the top of this file.

Note: the meaning of variable "Goal" and other variables ending in
"Goal" varies with context.  In some places they indicate an ICL
goal Source:Goal::Params (where Source and Params are both optional);
in other places, they indicate just the Goal part of an ICL goal.

\*/


execute_goal(RKS, OrigGoal, OrigParams, ICLGoal) :-
    % Here, ICLGoal may or may not include a Sources component.  Either
    % way, it gets handled by execute/7.
    % @@ What if OrigGoal's Params or GlobalParams has vars?
    % We remove addresses before calling term_vars only so as to avoid
    % a syntax error exception that comes up when ICLGoal = Addr:\+Goal
    remove_addresses(ICLGoal, TempGoal),
    term_vars(TempGoal, AllVars, _Singletons, _NonSingletons),
    new_goal_id(Id),
    % This means simply, "When the Solvers and solutions (in the form of
    % Bindings for AllVars) are known for Goal, call
    % unify_and_return_solutions(...)."
    assert(continuation(Id, Requestees, Solvers, Bindings,
           unify_and_return_solutions(Id,RKS,OrigGoal,OrigParams,AllVars,
                                      Requestees,Solvers,Bindings))),
    % This means: Find the Solvers and solutions:
    execute(Id, RKS, [], [], ICLGoal, OrigParams, AllVars).

/*\
 *     execute(Id, RKS, Requestees, Solvers, Goal, InheritedParams, Vars).

execute/7 satisfies the ICL goal Goal.  Id is an integer that
identifies a continuation assertion.  When the satisfaction of Goal
has been completed, the continuation assertion tells what to do next.
The satisfaction of Goal may be very simple, or may involve a number
of steps, depending on the form of Goal.

Requestees is a list of source id's of all sources asked to
participate in the satisfaction of whatever request contained Goal,
and Solvers is a list of source id's of sources that succeeded in
satisfying some part of the request (so Solvers is a subset of
Requestees.  These lists are being accumulated for return to the agent
that submitted the request.

Conceptually, execute/7 does this:
```

```
    findall(Vars, Goal, Bindings),
    append(Requestees, <list of KSs called on in the findall>, NewRequestees),
    append(Solvers, <list of KSs providing solutions in the findall>,
           NewSolvers),
    continue_execution(Id, RKS, NewRequestees, NewSolvers, Bindings)

The behavior of continue_execution, then, depends on a continuation/5
assertion, with Id as the first arg.

The important details have to do with how the satisfaction of the
"findall" part of this strategy may be delayed.

 *
\*/

execute(Id, RKS, Requestees, Solvers, built_in:ICLGoal, InheritedParams, Vars)
:-
    % This handles ICL built-ins, such as <, >, =, member/2, true, false, ...
    !,
    icl_GoalComponents(ICLGoal, _, Goal, Params),
    append(Params, InheritedParams, AllParams),
    oaa_Name(Facilitator),
    add_element(Facilitator, Requestees, NewRequestees),
    % If the requestor wants to know the solver, bind it here:
    ( memberchk(get_address(Facilitator), Params) -> true | true),

    ( oaa:passes_tests(Params) ->
        % @@The use of solution_limit and elsewhere here needs a close look:
        ( memberchk(solution_limit(N), AllParams) ->
          oaa:findNSolutions(N, Vars, call(Goal), Bindings)
      | otherwise ->
          findall(Vars, call(Goal), Bindings)
      )
    | otherwise ->
        Bindings = []
    ),
    ( Bindings == [] ->
        NewSolvers = Solvers
    | otherwise ->
        add_element(Facilitator, Solvers, NewSolvers)
    ),
    ( memberchk(reply(none), AllParams) ->
        continue_execution(Id, RKS, NewRequestees, NewSolvers, [Vars])
    | otherwise ->
        continue_execution(Id, RKS, NewRequestees, NewSolvers, Bindings)
    ).

    % Empty list of sources:
execute(Id, RKS, Requestees, Solvers, []:ICLGoal, _InheritedParams, _Vars) :-
    format('WARNING: No solvers for ICL goal or subgoal:~n  ~q~n',
         ICLGoal),
    continue_execution(Id, RKS, Requestees, Solvers, []).

    % Single KS in a list:
execute(Id, RKS, Requestees, Solvers, [KS]:G, Params, Vars) :-
    !,
```

```
    execute(Id, RKS, Requestees, Solvers, KS:G, Params, Vars).

    % Multiple KSs in a list:
execute(Id, RKS, Requestees, Solvers, [KS | Rest]:G, Params, Vars) :-
    !,
    execute_for_each_ks(Id, RKS, Requestees, Solvers, G, Params,
                   Vars, [KS | Rest]).

    % Solver is facilitator (me):
execute(Id, RKS, Requestees, Solvers, Source:ICLGoal, InheritedParams, Vars) :-
    oaa_Name(Facilitator),
    (Source = facilitator ; Source = Facilitator),
    !,
    icl_GoalComponents(ICLGoal, _, Goal, Params),
    % If the requestor wants to know the solver, bind it here:
    ( memberchk(get_address(Facilitator), Params) -> true | true),
    append(Params, InheritedParams, AllParams),
    findall(Vars,
            oaa:oaa_solve_local(Goal, InheritedParams),
          Bindings),
    ( memberchk(reply(none), AllParams) ->
        true
    | otherwise ->
      oaa_Name(KSName),
      add_element(KSName, Requestees, NewRequestees),
      ( Bindings == [] ->
            NewSolvers = Solvers
      | otherwise ->
            add_element(KSName, Solvers, NewSolvers)
      ),
      continue_execution(Id, RKS, NewRequestees, NewSolvers, Bindings)
    ).

% Note: this code was inherited from pre-compound-query facilitator.
% One significant change: when a goal is sent to a parent, we used to
% automatically include local blackboard solutions also.  We don't
% do this anymore.
%
% @@ Strategy should be re-evaluated at some point.  For instance,
% the use of var P2 might now cause things to break (the requesting
% agent might try to unify its copy of Params with P2).

execute(Id, RKS, Requestees, Solvers, Sources:ICLGoal, InheritedParams, Vars) :-
    on_exception(_, com:com_GetInfo(parent, fac_id(ParentBB)), fail),
    (Sources == parent ; Sources == ParentBB),
    !,

    icl_GoalComponents(ICLGoal, _, _Goal, Params),
    % If the requestor wants to know the solver, bind it here:
    % NO - it gets bound by the parent facilitator.
    % ( memberchk(get_address(ParentBB), Params) -> true | true),

    append(Params, InheritedParams, AllParams),
    % We don't need to check the level here (that's already been done),
    % but we do need to decrement its value by 1:
    find_level(AllParams, _Level, NewParams),
    oaa_TraceMsg('~nRouting goal "solve(~p)" to parent ~p.~n',
```

```
    [ICLGoal, ParentBB]),
    new_goal_id(NewId),
    oaa_PostEvent(ev_post_solve_from_bb(NewId, ICLGoal, NewParams),
                  [address(ParentBB)]),
    ( memberchk(reply(none), NewParams) ->
        unify_and_continue_execution(Id, RKS, ICLGoal, Vars,
          ParentBB, Requestees, Solvers, [ICLGoal])
    | otherwise ->
      % @@Shouldn't there be a time-check here?
      oaa:oaa_add_trigger_local(
              comm,
              event(ev_reply_solved_by_bb(NewId, _KS, ICLGoal, _P2,
                                          Solutions),
                  _),
            ev_unify_and_continue_execution(Id, RKS, ICLGoal, Vars,
                      ParentBB, Requestees, Solvers, Solutions),
              [recurrence(when), on(receive)])
    ).

    % Send the goal to an agent:
execute(Id, RKS, Requestees, Solvers, KS:ICLGoal, InheritedParams, Vars) :-
    !,
    icl_GoalComponents(ICLGoal, _, Goal, Params),
    append(Params, InheritedParams, AllParams),
    % @@What if the KS' status has changed since it was specified?
    % find_address allows for KS to be either numeric or symbolic.
    find_address(KS, KSId),
    % If the requestor wants to know the solver, bind it here:
    ( memberchk(get_address(KSId), Params) -> true | true),
    % Could do another check of the agent's validity:
    % ks_ready(KSId, _),

%     relevant_vars(Vars, Goal, GVars),
%     OptimizedG = findall(GVars, Goal, All),

    % Output trace message:
    ( oaa:oaa_trace(on) ->
        copy_term(ICLGoal, TraceCopy),
      numbervars(TraceCopy, 0, _),
        copy_term(InheritedParams, ParamsCopy),
      numbervars(ParamsCopy, 0, _),
      oaa_TraceMsg(
        '% Routing goal to ~w:~n%   ~w ~w~n~n',
        [KS, TraceCopy, ParamsCopy])
    | otherwise ->
        true
    ),

    new_goal_id(NewId),
%     oaa_PostEvent(KS, RKS, solve(NewId, OptimizedG::Params, [])),
    oaa_PostEvent(ev_solve(NewId, ICLGoal, InheritedParams),
                  [from(RKS), address(KSId)]),

    ( memberchk(reply(none), AllParams) ->
        unify_and_continue_execution(Id, RKS, ICLGoal, Vars,
                          KSId, Requestees, Solvers, [ICLGoal])
         % If time_limit specified in parameters, setup
```

```
        % time_trigger to wakeup if solutions hasn't been returned
        % in specified time.
    | otherwise ->
       ( memberchk(time_limit(NSecs), AllParams) ->
           add_time_check(NSecs, NewId, RKS, Goal,AllParams)
      | true),
      oaa:oaa_add_trigger_local(
        comm,
        event(ev_solved(NewId, _KS, ICLGoal, _P2, Solutions), _),
        ev_unify_and_continue_execution(Id, RKS, ICLGoal, Vars,
                                     KSId, Requestees, Solvers, Solutions),
        [recurrence(when), on(receive)])
%     poll_until_all_events([solved(Id, _KS, OptimizedG, P2, Solutions)]),
%     Solutions = [findall(GVars, Goal, All)],
%     respond_query(Id, RKS, Solvers, KS, Goal, P2, Solutions)
    % Backtrack over solutions:
%     member(GVars, All).
    ).

    % Negation:
execute(Id, RKS, Requestees, Solvers, ICLGoal, InheritedParams, Vars) :-
    icl_GoalComponents(ICLGoal, _, (\+ G1), Params),
    !,
    append(Params, InheritedParams, NewIParams),
    new_goal_id(NewId),
    assert(
        continuation(NewId, NewRequestees, NewSolvers, Bindings,
          continue_negation(Id, RKS, NewRequestees, NewSolvers, NewIParams,
                      Vars, Bindings))),
    execute(NewId, RKS, Requestees, Solvers, G1, NewIParams, Vars).

    % Conjunction:
execute(Id, RKS, Requestees, Solvers, ICLGoal, InheritedParams, Vars) :-
    icl_GoalComponents(ICLGoal, _, (G1, G2), Params),
    !,
    append(Params, InheritedParams, NewIParams),
    new_goal_id(NewId),
    assert(
        continuation(NewId, NewRequestees, NewSolvers, Bindings,
          continue_conjunction(Id, RKS, NewRequestees, NewSolvers, G2,
NewIParams,
                           Vars, Bindings))),
    execute(NewId, RKS, Requestees, Solvers, G1, NewIParams, Vars).

    % Local cut with alternative.  Note: this clause must precede
    % that for disjunction.
execute(Id, RKS, Requestees, Solvers, ICLGoal, InheritedParams, Vars) :-
    icl_GoalComponents(ICLGoal, _, (G1 -> G2 | G3), Params),
    !,
    append(Params, InheritedParams, NewIParams),
    new_goal_id(NewId),
    assert(
        continuation(NewId, NewRequestees, NewSolvers, Bindings,
          continue_local_cut(Id, RKS, NewRequestees, NewSolvers, G2, G3,
NewIParams,
                       Vars, Bindings))),
    execute(NewId, RKS, Requestees, Solvers, G1, NewIParams, Vars).
```

```
    % Local cut:
execute(Id, RKS, Requestees, Solvers, ICLGoal, InheritedParams, Vars) :-
    icl_GoalComponents(ICLGoal, _, (G1 -> G2), Params),
    !,
    append(Params, InheritedParams, NewIParams),
    new_goal_id(NewId),
    assert(
        continuation(NewId, NewRequestees, NewSolvers, Bindings,
          continue_local_cut(Id, RKS, NewRequestees, NewSolvers, G2, false,
NewIParams,
                         Vars, Bindings))),
    execute(NewId, RKS, Requestees, Solvers, G1, NewIParams, Vars).

    % Disjunction:
execute(Id, RKS, Requestees, Solvers, ICLGoal, InheritedParams, Vars) :-
    icl_GoalComponents(ICLGoal, _, (G1; G2), Params),
    !,
    append(Params, InheritedParams, NewIParams),
    new_goal_id(Id1),
    new_goal_id(Id2),
    assert(
        multiple_continuation([Id1, Id2], Requestees, AllRequestees,
                        Solvers, AllSolvers,
                        [], AllBindings,
          continue_execution(Id, RKS, AllRequestees, AllSolvers, AllBindings))),
    execute(Id1, RKS, Requestees, Solvers, G1, NewIParams, Vars),
    execute(Id2, RKS, Requestees, Solvers, G2, NewIParams, Vars).

    % Occasionally, a goal may have the form G::P (that is, no
    % address, and P is not compound), but it is still valid, so
    % long as G is valid.
    %
    % Ex.: ([7]:a1(1)::[...])::[...]
execute(Id, RKS, Requestees, Solvers, Goal::Params, InheritedParams, Vars) :-
    !,
    append(Params, InheritedParams, NewIParams),
    execute(Id, RKS, Requestees, Solvers, Goal, NewIParams, Vars).

execute(Id, RKS, Requestees, Solvers, G, _Params, _Vars) :-
    format('WARNING (execute/7): unrecognized goal form:~n    ~w~n', [G]),
    continue_execution(Id, RKS, Requestees, Solvers, []).

execute_for_each_ks(Id, RKS, Requestees, Solvers, Goal, Params, Vars, KSs) :-
    length(KSs, NumKSs),
    new_goal_ids(NumKSs, Ids),
    assert(
        multiple_continuation(Ids, Requestees, AllRequestees, Solvers,
AllSolvers, [], AllBindings,
          continue_execution(Id, RKS, AllRequestees, AllSolvers, AllBindings))),
    exec_for_each_ks(NumKSs, Ids, KSs, RKS, Requestees, Solvers, Goal,
                Params, Vars).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GOAL EXECUTION: INTERMEDIATE STEPS
% The predicates in this group define intermediate steps in the satisfaction
% of various ICL goal forms.
```

```
%
% Note: intermediate steps in handling of DISJUNCTION are handled by
% continue_execution, using the multiple_continuation assertion.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % This is used in satisfying [KS1, KS2, ...]:Goal.  Note that this is
    % equivalent to a disjunction (KS1:Goal ; KS2:Goal ; ....).  So we
    % are able to use the multiple_continuation assertion to accumulate
    % the solutions.
    %
    % We don't need Solvers, because ...

exec_for_each_ks(NumKSs, Ids, KSs, RKS, _Requestees, _Solvers,
             Goal, Params, Vars) :-
    retractall( ks_num(_) ),
    assert( ks_num(1) ),
    repeat,
    ks_num(Num),
    ( Num > NumKSs ->
      !
    | otherwise ->
      nth1(Num, KSs, KS),
      nth1(Num, Ids, Id),
      % We use a local cut to prevent some (harmless) backtracking.
      % This is one place where we don't need to pass Requestees and
      % Solvers through to execute (3rd and 4th args), because they are
      % filled in by handle_multiple_continuation.

      ( execute(Id, RKS, [], [], KS:Goal, Params, Vars) -> true ),
      NextNum is Num + 1,
      retractall( ks_num(_) ),
      assert( ks_num(NextNum) ),
      fail
    ).

    % This is used in satisfying (\+ Goal).  When this
    % pred. is called, Goal has just been completed.  Bindings gives
    % the solutions to  Goal.

continue_negation(Id, RKS, Requestees, Solvers, _Params, Vars, []) :-
    !,
    continue_execution(Id, RKS, Requestees, Solvers, [Vars]).
continue_negation(Id, RKS, Requestees, Solvers, _Params, _Vars, _Bindings) :-
    continue_execution(Id, RKS, Requestees, Solvers, []).

    % This is used in satisfying (Goal1, Goal2).  When this
    % pred. is called, Goal1 has just been completed.  Bindings gives
    % the solutions to Goal1.

continue_conjunction(Id, RKS, Requestees, Solvers, _Goal2, _Params, _Vars, [])
:-
    !,
    continue_execution(Id, RKS, Requestees, Solvers, []).
continue_conjunction(Id, RKS, Requestees, Solvers, Goal2, Params, Vars,
Bindings) :-
    length(Bindings, NumBindings),
    new_goal_ids(NumBindings, Ids),
```

```
    assert(
        multiple_continuation(Ids, Requestees, AllRequestees, Solvers,
AllSolvers, [], AllBindings,
          continue_execution(Id, RKS, AllRequestees, AllSolvers, AllBindings))),
    exec_for_each_binding(NumBindings, Ids, Bindings, RKS, Requestees, Solvers,
Goal2,
              Params, Vars).

    % We don't need Requestees or Solvers, because they are filled in
    % by handle_multiple_continuation.

exec_for_each_binding(NumBindings, Ids, Bindings, RKS, _Requestees, _Solvers,
            Goal, Params, Vars) :-
    retractall( binding_num(_) ),
    assert( binding_num(1) ),
    repeat,
    binding_num(Num),
    ( Num > NumBindings ->
      !
    | otherwise ->
      nth1(Num, Bindings, Binding),
      nth1(Num, Ids, Id),
      Vars = Binding,
      % We use a local cut to prevent some (harmless) backtracking.
      % This is one place where we don't need to pass Solvers through
      % to execute (3rd arg):
      ( execute(Id, RKS, [], [], Goal, Params, Binding) -> true ),
      NextNum is Num + 1,
      retractall( binding_num(_) ),
      assert( binding_num(NextNum) ),
      fail
    ).

    % This is used in satisfying Goal1 -> Goal2 | Goal3.  When this
    % pred. is called, Goal1 has just been completed.  Bindings gives
    % the solutions to Goal1.

    % No solutions to Goal1:
continue_local_cut(Id, RKS, Requestees, Solvers, _Goal2, Goal3, Params,
               Vars, []) :-
    !,
    ( Goal3 = false ->
        continue_execution(Id, RKS, Requestees, Solvers, [])
    | otherwise ->
      execute(Id, RKS, Requestees, Solvers, Goal3, Params, Vars)
    ).
    % Some solutions:
continue_local_cut(Id, RKS, Requestees, Solvers, Goal2, _Goal3, Params,
               Vars, [Binding1 | _]) :-
    new_goal_id(NewId),
    assert(
        continuation(NewId, NewRequestees, NewSolvers, Bindings,
          continue_execution(Id, RKS, NewRequestees, NewSolvers, Bindings))),
    Vars = Binding1,
    % local cut to prevent some (harmless) backtracking:
    ( execute(NewId, RKS, Requestees, Solvers, Goal2, Params, Binding1) -> true
).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GOAL EXECUTION: COMPLETION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This is called when the goal associated with Id has been completely
% satisfied.

continue_execution(Id, _RKS, Requestees, Solvers, Bindings) :-
      % Here we are BINDING the Solvers and Bindings var's. in the
      % continuation assertion.  The var. also appears in Continuation:
    ( retract(continuation(Id, Requestees, Solvers, Bindings, Continuation)) ->
      call(Continuation)
    | multiple_continuation(Ids, _, _, _, _, _, _),
      memberchk(Id, Ids) ->
      handle_multiple_continuation(Id, Requestees, Solvers, Bindings, Ids)
    | otherwise ->
      format('Internal Error: no continuation with id ~w~n', [Id])
    ).

handle_multiple_continuation(Id, Requestees, Solvers, Bindings, Ids) :-
    retract(multiple_continuation(Ids, PrevRequestees,
                      AllRequestees, PrevSolvers, AllSolvers,
                      PrevBindings, AllBindings,
                      Continuation)),
    del_element(Id, Ids, NewIds),
    append(PrevBindings, Bindings, NewBindings),
    append(PrevRequestees, Requestees, NewRequestees),
    append(PrevSolvers, Solvers, NewSolvers),
    ( NewIds = [] ->
      AllBindings = NewBindings,
      AllRequestees = NewRequestees,
      AllSolvers = NewSolvers,
      call(Continuation)
    | otherwise ->
      assert(multiple_continuation(NewIds, NewRequestees, AllRequestees,
                        NewSolvers, AllSolvers,
                        NewBindings, AllBindings,
                        Continuation))
    ).

% @@Let's see, if these args included the vars for any
% nested solvers params, we could probably instantiate solvers
% params in Goal...

unify_and_continue_execution(Id, RKS, Goal, Vars, Requestee, Requestees,
                       Solvers, Solutions) :-
    add_element(Requestee, Requestees, NewRequestees),
    ( Solutions == [] ->
        NewSolvers = Solvers
    | otherwise ->
        add_element(Requestee, Solvers, NewSolvers)
    ),
    findall(Vars,
          member(Goal, Solutions),
          Bindings),
    continue_execution(Id, RKS, NewRequestees, NewSolvers, Bindings).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GENERAL UTILITIES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

term_vars(Term, AllVars, SingletonVars, NonSingletonVars) :-
    with_output_to_chars(portray_clause(Term), Chars),
    with_input_from_chars(
                read_term([variable_names(Names), singletons(Singletons)],
                    Term1),
                Chars),
    extract_vars(Names, Singletons, AllVars, SingletonVars, NonSingletonVars),
    Term = Term1.

extract_vars([], _Singletons, [], [], []).
extract_vars([Name = Var | RestNames], Singletons, [Var | RestVars],
           [Var | RestSV], NonSingletonVars) :-
    memberchk(Name = Var, Singletons),
    !,
    extract_vars(RestNames, Singletons, RestVars, RestSV, NonSingletonVars).
extract_vars([_Name = Var | RestNames], Singletons, [Var | RestVars],
           RestSV, [Var | NonSingletonVars]) :-
    extract_vars(RestNames, Singletons, RestVars, RestSV, NonSingletonVars).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEBUGGING UTILITIES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% static_test :-
%     Class = root,
%     KSName = dontcare,
%     BBName = dontcare,
%     oaa_read_setup_file,
%     oaa_init_flags,
%     assert(oaa_class(Class)),
%     oaa_SetupCommunication(Class, KSName, BBName, []),
%     on_exception(_, oaa_AppInit, true),
%     oaa_Ready(true).
%
% connect :-
%     % go(leaf, shell, root).
%     static_test.
%
% ce :-
%        repeat,
%        oaa_GetEvent(CallingKS, Event, 0),
%        ( Event = timeout ->
%        !,
%            format('No events~n', [])
%        | otherwise ->
%            oaa_process_event(CallingKS, Event),
%        fail
%        ).
% ce :-
%     format('No events~n', []).
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% OrigGoal must be used in the return event, so that the
% requesting KS will identify it correctly.

unify_and_return_solutions(Id,RKS,OrigGoal,OrigParams,Vars,Requestees,Solvers,Bi
ndings) :-
    findall(OrigGoal,
          member(Vars, Bindings),
          Solutions),
    oaa_TraceMsg('~nRouting answers back to ~p:~n   ~p~n',
            [RKS,Solutions]),
    cancel_time_check(Id),
    remove_dups(Requestees, RequesteesSet),
    remove_dups(Solvers, SolversSet),
    % If present, bind solvers request in OrigParams:
    ( memberchk(get_address(RequesteesSet), OrigParams) -> true | true ),
    ( memberchk(get_satisfiers(SolversSet), OrigParams) -> true | true ),
    oaa_PostEvent(ev_reply_solved(RequesteesSet, SolversSet, OrigGoal,
OrigParams, Solutions),
                  [address(RKS)]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```